Eigene Programmiersprache? WebAssembly macht's möglich!

Inwiefern erleichtert WebAssembly den Bau eigener Compiler für Amateurentwickler?"

Facharbeit im Seminarfach »Zukunft des Digitalen«

Vorgelegt von: **Erik Tschöpe**

Abgabedatum: 19. Februar 2026

Inhaltsverzeichnis

1 Einleitung

Unterschiedliche Programmiersprachen haben die verschiedensten Funktionen, ihre eigene Syntax oder sind speziell auf einen Anwendungsfall zugeschnitten. So ist es ein Traum vieler Entwickler sich ihre eigene Programmiersprache zu erschaffen die perfekt an die eigenen Bedürfnisse angepasst ist.

Trotzdem bleibt die Entwicklung einer eigenen Programmiersprache für viele Entwickler ein fernes Ziel, da sie häufig mit einem hohen technischen Aufwand verbunden ist und viel Erfahrung erfordert. Ein wesentlicher Grund dafür liegt im Bau eines sogenannten Compilers.

Ein Compiler ist ein Programm, welches Quellcode in eine für den Computer ausführbare Form übersetzt. Der komplexeste teil ist dabei das Backend, welches für die Erzeugung von plattformspezifischem Maschinencode verantwortlich ist. Unterschiedliche Prozessorarchitekturen und Betriebssysteme erfordern jeweils eigene Lösungen, was den Entwicklungsaufwand stark erhöht. Aus diesem Grund werden Compiler in der Regel von größeren Entwicklerteams oder Unternehmen realisiert und nur selten von einzelnen Amateurentwicklern.

Mit der Einführung von WebAssembly (WASM) im Jahr 2017 wurde ein neuer Ansatz vorgestellt, der genau an dieser Stelle ansetzt. WebAssembly stellt ein standardisiertes, plattformunabhängiges Ausführungsformat dar [1], das von modernen Webbrowsern und zunehmend auch außerhalb des Webs unterstützt wird. Statt direkt Maschinencode für eine bestimmte Architektur zu erzeugen, können Compiler WebAssembly als gemeinsames Ziel verwenden. Dadurch werden viele hardware- und betriebssystemspezifische Details abstrahiert.

Diese Entwicklung wirft die Frage auf, ob WebAssembly den Einstieg in den Compilerbau grundlegend erleichtert. Insbesondere stellt sich die Frage, ob es durch WebAssembly erstmals realistisch wird, dass auch Amateurentwickler eigene, funktionsfähige Compiler entwickeln können.

Aus diesem Zusammenhang ergibt sich die Leitfrage dieser Facharbeit: **Inwiefern erleichtert WebAssembly den Bau eigener Compiler für Amateurentwickler?**

Zur Beantwortung dieser Frage werden zunächst die grundlegenden Konzepte des Compilerbaus sowie die Funktionsweise von WebAssembly erläutert. Anschließend wird in einem praktischen Selbstversuch ein einfacher Mini-Compiler für eine eigens definierte Programmiersprache entwickelt, der WebAssembly-Bytecode generiert. Auf Basis dieser Erfahrungen werden die

Chancen und Grenzen von WebAssembly als Compilation Target für Hobby-Compiler bewertet.

Die Ergebnisse zeigen, dass WebAssembly in vielerlei Hinsicht den Einstieg in den Compilerbau erleichtert. Insbesondere die plattformunabhängige Natur von WASM und die Abstraktion von Hardwaredetails ermöglichen es Entwicklern, sich auf die Implementierung der Sprache und der Compiler-Logik zu konzentrieren, ohne sich um die spezifischen Anforderungen verschiedener Zielarchitekturen kümmern zu müssen.

2 Compiler

Was ist ein Compiler? Wie ist er aufgebaut? Welche Rolle spielt das Backend? Warum ist WebAssembly als Ziel interessant?

2.1 Was ist ein Compiler?

Einen Compiler ist ein Programm, welches Programmcode in eine andere für Computer verständliche Form übersetzt. Dabei ist es ganz egal, ob es Binärcode für eine bestimmte Prozessorarchitektur, Bytecode für eine Virtuelle Maschine oder eine Zwischenrepräsentation für die Weiterverarbeitung ist. In Abgrenzung zu einem Interpreter führt ein Compiler den Code nicht direkt aus, sondern übersetzt ihn nur und führt dabei optional Optimierungen durch. Kompilierte Programme laufen dadurch in der Regel schneller als interpretierte Programme, da die Übersetzung bereits vor der Ausführung stattfindet und Optimierungen vorgenommen werden können. Zusätzlich erleichtern moderne Compiler den Entwicklern das Leben, indem sie häufige Fehler schon beim Übersetzen des Quellcodes finden und verständliche Fehlermeldungen ausgeben, während Interpreter Fehler erst zur Laufzeit sichtbar werden, was die Fehlersuche erschwert [2]. Wenn der Begriff „Compiler“ fällt, ist selten nur der reine Übersetzungsvorgang gemeint, sondern oft die gesamte Toolchain, die auch Assembler und Linker umfasst, um aus Quellcode eine ausführbare Datei zu erzeugen [3].

2.2 Aufbau eines Compilers (Frontend, Backend)

Ein Compiler ist grundlegend in mehrere Teile unterteilt, die jeweils klar abgegrenzte Aufgaben übernehmen: Lexing (Erzeugung von Token aus Quelltext), Parsing (Aufbau eines abstrakten Syntaxbaums, AST), semantische Analyse (Typprüfung, Namensauflösung, Scope- und Fehlerprüfung), eine Zwischenrepräsentation und Optimierungsphase (IR-Transformationen, konstante Auswertung, Dead-Code-Elimination) sowie das Backend (Code- bzw. Bytecode-Generierung, z.B. für WebAssembly). Diese Modularität erleichtert Entwicklung, Testbarkeit und Wiederverwendbarkeit der einzelnen Komponenten. Zusätzlich bietet diese Struktur die Möglichkeit, verschiedene Frontends (für unterschiedliche Sprachen) mit demselben Backend zu kom-

binieren, was die Flexibilität erhöht. Moderne Compiler sind genau entlang solcher Schritte aufgebaut [4].

Zusätzliche Stichpunkte:

- In realen Toolchains werden Vorverarbeitung, Kompilierung, Assemblierung und Linking als getrennte Schritte modelliert [3].

Quellen: [2]; [4]; [5]

2.3 Einführung in WebAssembly

WebAssembly (WASM) ist ein binäres, plattformunabhängiges Ausführungsformat, das ursprünglich für die Ausführung in Webbrowsern entwickelt wurde, aber inzwischen auch außerhalb des Webs, z.B. auf Servern oder in Tools, genutzt werden kann. Es zielt darauf ab, eine nahe an nativer Geschwindigkeit liegende Ausführung zu ermöglichen, während es gleichzeitig portabel und sicher bleibt. WASM-Module bestehen aus Funktionen, Speicher, Tabellen sowie Import- und Exportdefinitionen. Die Ausführung erfolgt in einer Sandbox-Umgebung, wobei der Zugriff auf Systemfunktionen über Host-Imports erfolgt. WASM wird von vielen Sprachen unterstützt, darunter C/C++, Rust und AssemblyScript, die über Compiler-Toolchains in WASM übersetzt werden können. Für den Compilerbau bietet WASM eine einheitliche Zielplattform, wodurch viele plattformspezifische Details im Backend entfallen. Unter anderem deswegen ist WASM besonders attraktiv für Hobby-Compiler, da es die Komplexität der Codegenerierung reduziert und den Fokus auf die Sprachlogik und -semantik ermöglicht [6].

Das Grundprinzip nach dem Wasm arbeitet ist die Stack-Maschine, bei der Instruktionen primär auf einem Operand-Stack operieren. Zum Beispiel nimmt die `add`-Instruktion die obersten zwei Werte vom Stack, addiert sie und legt das Ergebnis wieder auf den Stack. Dies ermöglicht eine einfache und effiziente Ausführung von Anweisungen, da keine expliziten Register oder Speicheradressen benötigt werden [7]; [1].

- Module werden vor der Ausführung validiert (z.B. Typkonsistenz von Instruktionen) [8].
- Textformat (WAT) und Binärformat bilden dieselbe Modulstruktur ab; WAT ist vor allem für Debugging und Lernen nützlich [7]; [1].

Hier ein Beispiel von Quellcode in unserer eigenen Sprache, der eine einfache Addition durchführt und das entsprechende WebAssembly Textformat (WAT), das daraus generiert wird.

```
fn add(a, b) -> Int {
    return a + b;
}

fn main(){
    print(add(7, 5));
    return;
}
```

Codebeispiel 1: Programmbeispiel: Addition in eigener Sprache (Quelle: add.eres)

```
(module
  (type (;0;) (func (param i64)))  // Funktion mit 1 Parameter, kein
Rückgabewert
  (type (;1;) (func (param i64 i64) (result i64)))  // Funktion mit 2 Parametern
und Rückgabe
  (type (;2;) (func))  // Funktion ohne Parameter, ohne Rückgabe
  (import "env" "print_i64" (func (;0;) (type 0)))  // Import: print_i64 aus env
  (export "add" (func 1))  // Export: Funktion 1 heißt add
  (export "main" (func 2))  // Export: Funktion 2 heißt main

  (func (;1;) (type 1) (param i64 i64) (result i64)  // Funktion add(a,b) -> i64
    local.get 0  // lade Parameter a
    local.get 1  // lade Parameter b
    i64.add  // addiere a + b
    return  // Ergebnis zurückgeben
    i64.const 0  // (Default-Return, falls kein return)
    return
  )

  (func (;2;) (type 2)  // Funktion main()
    i64.const 7  // konstante 7 auf den Stack
    i64.const 5  // konstante 5 auf den Stack
    call 1  // rufe add(7,5) auf
    call 0  // rufe print_i64(result) auf
    return  // Ende
  )
)
```

Codebeispiel 2: Generiertes WAT zum Additionsbeispiel (aus *add.eres*)

3 Selbstversuch

- Ziel: Umsetzbarkeit der Theorie im eigenen Mini-Compiler prüfen
- Fokus: vollständige Pipeline von Quelltext bis Ausführung
- Ergebnisartefakte: Tokens, AST, WAT, Laufzeitausgabe

Die Konzepte des Compilerbaus und die Funktionsweise von WebAssembly wurden nun theoretisch erläutert. Um die praktische Umsetzbarkeit dieser Konzepte zu überprüfen, wird im folgenden Abschnitt ein eigener Mini-Compiler entwickelt. Dieser Compiler soll eine eigens definierte, minimalistische Programmiersprache in WebAssembly-Bytecode übersetzen. Dabei wird die gesamte Pipeline von der Quelltexteingabe über die Tokenisierung, das Parsing, die semantische Analyse bis hin zur Codegenerierung und Ausführung durchlaufen. Ziel ist es, nicht nur die technischen Schritte zu demonstrieren, sondern auch konkrete Artefakte wie die erzeugten Tokens, den abstrakten Syntaxbaum (AST), das generierte WAT und die Laufzeitausgabe zu präsentieren.

3.1 Funktionsumfang der eigenen Programmiersprache

Diese Sprache ist bewusst minimalistisch gehalten, um den Fokus auf die Kernkonzepte des Compilerbaus zu legen. Sie unterstützt nur einen Datentyp (`Int`), der intern als 64-Bit-Ganzzahl (`i64`) umgesetzt wird. Es gibt keine globalen Variablen, sondern nur Funktionen, die lokale Variablen über `let`-Statements definieren können. Kontrollstrukturen umfassen `if/else` und `while`, während Ausdrücke Literale, Variablen, Binäroperationen und Funktionsaufrufe erlauben. Vergleichsoperatoren ermöglichen einfache Bedingungen. Rückgabetypen sind optional, und eine Host-Funktion `print` ermöglicht die Ausgabe von Werten.

```rust
#[derive(Debug)]
pub enum Stmt {
    Let { name: String, value: Expr },
    Return(Option<Expr>),
    Expr(Expr),
    If { cond: Expr, then_block: Vec<Stmt>, else_block: Vec<Stmt> },
    While { cond: Expr, body: Vec<Stmt> },
}

#[derive(Debug)]
pub struct FunctionDecl {
    pub name: String,
    pub params: Vec<String>,
    pub body: Vec<Stmt>,
    pub return_type: Option<Type>,
}
```

Codebeispiel 3: AST-Datenstrukturen (Quelle: src/ast.rs)

Erläuterung:

- `Stmt` beschreibt die möglichen Anweisungen der Sprache.
- `FunctionDecl` hält Signatur und Funktionskörper zusammen.

3.2 Lexer / Scanner

Der Lexer liest den Quelltext Zeichen für Zeichen und gruppiert sie in sinnvolle Einheiten, sogenannte Tokens. Er erkennt Schlüsselwörter wie `fn`, `let`, `if`, `else`, `while` und `return`, die eine spezielle Bedeutung haben. Außerdem identifiziert er Literale (z.B. Ganzzahlen) und Identifier (z.B. Funktions- oder Variablennamen). Operatoren und Trennzeichen werden ebenfalls als eigene Token klassifiziert. Bei der Verarbeitung des Quelltexts muss der Lexer auch Fehler erkennen, z.B. wenn ein unerwartetes Zeichen auftaucht oder eine Zahl ungültig formatiert ist [9]; [5].

Priorisierte Stichpunkte (Lexer):
- [MUSS] Klarer Ablauf: Whitespace überspringen -> Zahl/Identifier/Operator erkennen -> Token ausgeben.
- [MUSS] Trennlogik erklären: Ein Token endet, sobald ein Zeichen nicht mehr zur aktuellen Klasse passt.
- [MUSS] Schlüsselwort vs. Identifier erklären: gleiche Lesephase, Entscheidung erst am Ende.
- [MUSS] Fehlerfall erklären: unbekanntes Zeichen erzeugt direkt einen Lexer-Fehler.
- [NICE] Typische Mini-Beispiele angeben: `let x=3;` -> `Let, Ident(x), Equal, Int(3), Semicolon`.
- [STREICHEN] Vollständige Auflistung jedes einzelnen Tokens im Fließtext.

```rust
fn next_token(chars: &mut std::iter::Peekable<std::str::Chars<'_>>) -> TokenKind
{
    while let Some(&c) = chars.peek() {
        if c.is_whitespace() {
            chars.next();
            continue;
        }

        if c.is_ascii_alphabetic() || c == '_' {
            return lex_ident(chars);
        }
        if c.is_ascii_digit() {
            return lex_number(chars);
        }

        chars.next();
        return match c {
            '+' => TokenKind::Plus,
            '-' => TokenKind::Minus,
            ';' => TokenKind::Semicolon,
            _ => TokenKind::Error,
        };
    }

    TokenKind::EOF
}
```

Codebeispiel 4: Vereinfachter Lexer-Hauptlauf (Quelle: src/lexer.rs)

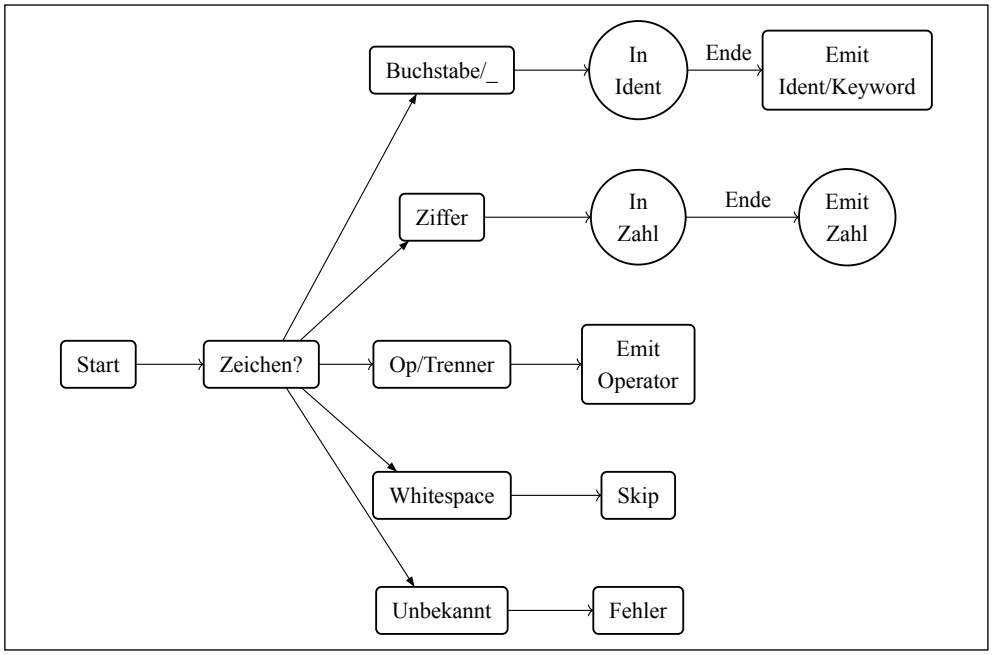

Abbildung 1: Zustandsmodell des Lexers

```rust
pub enum TokenKind {
    // Schlüsselwörter
    Let, Return, If, While,

    // Inhalte
    Ident(String), Int(i64),

    // Operatoren / Trenner
    Plus, Minus, Star, Slash, Equal,
    LParen, RParen, LBrace, RBrace, Semicolon,

    // Ende / Fehler
    EOF,
    Error,
}
```

Codebeispiel 5: Token-Typen (Quelle: src/token.rs)

Erläuterung:

- Die Tokenliste ist die gemeinsame Sprache zwischen Lexer und Parser.
- `Ident(String)` und `Int(i64)` tragen bereits konkrete Werte.

```rust
fn keyword_or_ident(text: String) -> TokenKind {
    match text.as_str() {
        "let" => TokenKind::Let,
        "return" => TokenKind::Return,
        "if" => TokenKind::If,
        "while" => TokenKind::While,
        _ => TokenKind::Ident(text),
    }
}

fn lex_ident(chars: &mut std::iter::Peekable<std::str::Chars<'_>>) -> TokenKind
{
    let mut text = String::new();
    while let Some(&c) = chars.peek() {
        if c.is_ascii_alphanumeric() || c == '_' {
            text.push(c);
            chars.next();
        } else {
            break;
        }
    }
    keyword_or_ident(text)
}
```

Codebeispiel 6: Identifier-Lexing (Quelle: src/lexer.rs)

Erläuterung:

- Zeichenfolge wird gesammelt und dann gegen Schlüsselwörter geprüft.
- Alles, was kein Schlüsselwort ist, wird als `Ident(...)` behandelt.

```rust
fn lex_number(chars: &mut std::iter::Peekable<std::str::Chars<'_>>) -> TokenKind
{
    let mut text = String::new();
    while let Some(&c) = chars.peek() {
        if c.is_ascii_digit() {
            text.push(c);
            chars.next();
        } else {
            break;
        }
    }

    match text.parse::<i64>() {
        Ok(value) => TokenKind::Int(value),
        Err(_) => TokenKind::Error,
    }
}
```

Codebeispiel 7: Zahlen-Lexing (vereinfacht, Quelle: src/lexer.rs)

3.3 Parser

Der Parser nimmt die vom Lexer erzeugte Tokenliste und baut daraus einen abstrakten Syntaxbaum (AST) auf, der die hierarchische Struktur des Programms widerspiegelt. Der Einstiegspunkt ist die Funktion `parse_program`, die alle Funktionen im Quelltext sammelt, bis sie das End-Token (`EOF`) erreicht. Jede Funktion wird durch `fn name(params) -> Int { ... }` definiert, wobei der Rückgabetyp optional ist. Blöcke werden als Sequenzen von Statements in `{ ... }` dargestellt. Für Ausdrücke wird ein spezieller Parser mit Präzedenzregeln implementiert, um die korrekte Bindung von Operatoren sicherzustellen.

Zusätzliche Stichpunkte:

- Der gewählte Ansatz entspricht einem rekursiven Abstieg, bei dem Nichtterminale durch Funktionen umgesetzt werden [10].
- Operator-Präzedenz wird typischerweise über eine Prioritätstabelle gesteuert, damit z.B. * stärker bindet als + [10].
- Der AST trennt konkrete Syntax (Tokens, Klammern) von semantisch relevanter Struktur (Ausdrücke, Statements) [5].
- [MUSS] Operator-Präzedenz: `*` und `/` werden vor `+` und `-` gebunden.
- [MUSS] Beispiel zur Präzedenz: `1 + 2 * 3` ergibt AST-Form `Add(1, Mul(2, 3))`.
- [MUSS] Assoziativität: `10 - 3 - 2` wird als `(10 - 3) - 2` geparst (linksassoziativ).
- [MUSS] Fehlerstrategie: bei unerwartetem Token sofort `Err(...)`, kein Weiterparsen.
- [MUSS] Ablauf `parse_expr`: zuerst linker Basis-Ausdruck, dann Operatoren in Präzedenz-Reihenfolge anhängen.
- [NICE] Mini-Beispiel AST-Form: Quelle `a + b * c` -> `Add(Var(a), Mul(Var(b), Var(c)))`.
- [NICE] Scope-Verhalten bei Blöcken benennen: Variablenzuordnung über lokale Indizes statt echter Symboltabellen-Hierarchie.
- [NICE] Optional 1 Tabelle mit Operatoren + Priorität + Assoziativität.
- [STREICHEN] sehr detaillierte Randfälle (z.B. alle nicht unterstützten Syntaxformen) nur kurz nennen statt breit ausführen.

Umsetzung der [MUSS]-Punkte (stichpunktartig):

- Präzedenz wird als Zahl modelliert (* > +).
- Die `while`-Schleife in `parse_expr` hängt so lange Operatoren an, wie deren Präzedenz hoch genug ist.
- Linksassoziativität entsteht durch `parse_expr(prec + 1)` auf der rechten Seite.

- Fehler entstehen zentral über `expect(...)` und werden als `Result::Err` weitergegeben.

```rust
fn precedence(kind: &TokenKind) -> Option<u8> {
    match kind {
        TokenKind::Star | TokenKind::Slash => Some(20),
        TokenKind::Plus | TokenKind::Minus => Some(10),
        _ => None,
    }
}

fn parse_expr(&mut self, min_prec: u8) -> Result<Expr, ParseError> {
    let mut left = self.parse_primary()?;

    while let Some(op_prec) = precedence(&self.peek().kind) {
        if op_prec < min_prec {
            break;
        }

        let op = self.bump().kind.clone();
        let right = self.parse_expr(op_prec + 1)?; // linksassoziativ
        left = Expr::Binary(Box::new(left), op, Box::new(right));
    }

    Ok(left)
}
```

Codebeispiel 8: Operator-Präzedenz (vereinfacht, Quelle: src/parser.rs)

```rust
fn parse_stmt(&mut self) -> Result<Stmt, ParseError> {
    match self.peek().kind {
        TokenKind::Let => self.parse_let(),
        TokenKind::Return => self.parse_return(),
        TokenKind::If => self.parse_if(),
        TokenKind::While => self.parse_while(),
        _ => self.parse_expr_stmt(),
    }
}
```

Codebeispiel 9: Statement-Dispatch (vereinfacht, Quelle: src/parser.rs)

```rust
fn expect(&mut self, expected: TokenKind) -> Result<(), ParseError> {
    let found = self.bump().kind.clone();
    if found == expected {
        Ok(())
    } else {
        Err(ParseError::new(expected, found))
    }
}
```

Codebeispiel 10: Fehlerstrategie mit *expect* (vereinfacht, Quelle: src/parser.rs)

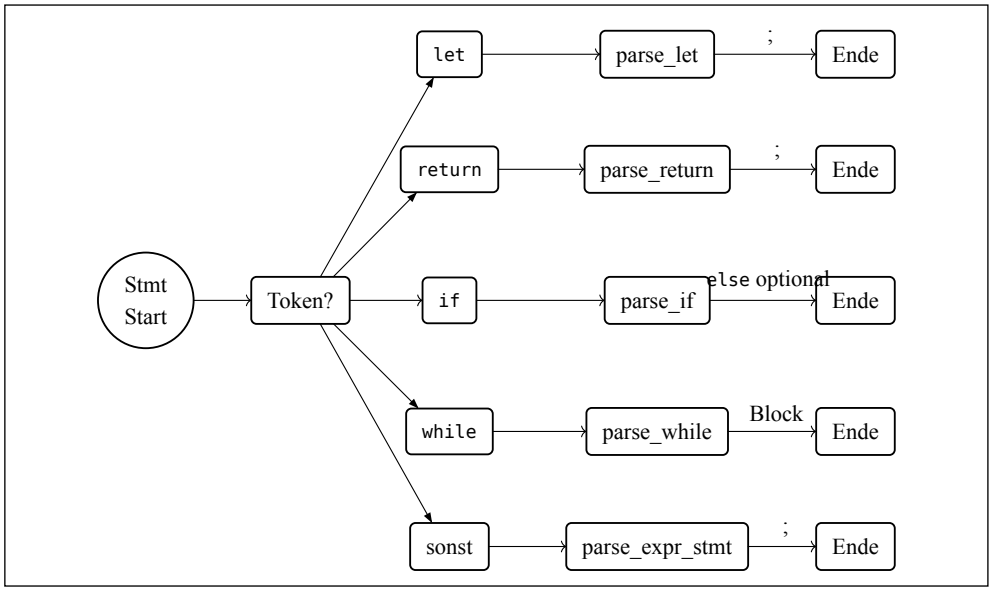

Abbildung 2: Zustandsmodell der Statement-Parserlogik

Hinweis: `parse_expr` arbeitet rekursiv (z.B. Klammern, Binäroperatoren).

```rust
pub fn parse_program(&mut self) -> Result<Program, ParseError> {
    let mut functions = Vec::new();

    while self.peek().kind != TokenKind::EOF {
        let func = self.parse_function()?;
        functions.push(func);
    }
    Ok(Program { functions })
}
```

Codebeispiel 11: Parser-Einstieg (*parse_program*) (Quelle: src/parser.rs)

Erläuterung:

- Der Parser sammelt alle Funktionen bis zum End-Token.
- Ergebnis ist ein `Program` als Einstiegsknoten des AST.

```rust
fn parse_function(&mut self) -> Result<FunctionDecl, ParseError> {
    self.expect(TokenKind::Fn)?;
    let name = self.expect_ident()?;
    self.expect(TokenKind::LParen)?;
```

Codebeispiel 12: Funktionsparser Teil 1 (Quelle: src/parser.rs)

Erläuterung:

- Erwartet `fn`, danach den Namen und die Parameterliste.
- Rückgabetyp ist optional und wird nur bei `-> Int` gesetzt.
- Der Funktionskörper ist ein Block mit Statements.

```rust
    let params = self.parse_params()?;
    self.expect(TokenKind::RParen)?;
```

Codebeispiel 13: Funktionsparser Teil 2 (Quelle: src/parser.rs)

```
    let return_type = if self.peek().kind == TokenKind::Arrow {
        self.bump();
        self.expect(TokenKind::IntType)?;
        Some(Type::Int)
    } else {
        None
    };

    let body = self.parse_block()?;

    Ok(FunctionDecl {
        name,
        params,
        body,
        return_type,
    })
}
```

Codebeispiel 14: Funktionsparser Teil 3 (Quelle: src/parser.rs)

3.4 Codegen

Der Codegenerator nimmt den AST und übersetzt ihn in eine eigene Zwischenrepräsentation (IR), die aus einer linearen Folge von Instruktionen besteht. Diese IR abstrahiert von den Details der WASM-Generierung und ermöglicht eine klarere Trennung zwischen der Logik der Codeerzeugung und den spezifischen Anforderungen des WASM-Formats. Anschließend wird die IR in echte WASM-Bytecode-Instruktionen umgewandelt, wobei die `wasm_encoder`-Bibliothek verwendet wird, um Module, Funktionen, Imports und Exports zu definieren. Die Host-Funktion `print` wird als Import unter dem Namen `env.print_i64` bereitgestellt, damit sie im generierten WASM-Modul aufgerufen werden kann.

Zusätzliche Stichpunkte:

- Eine IR erleichtert spätere Optimierungen, weil Transformationen nicht mehr direkt auf Quellsyntax arbeiten.
- Der Einsatz einer IR entspricht der Praxis großer Compiler-Infrastrukturen (z.B. LLVM IR als zentrale Zwischenschicht) [11].
- Die Abbildung von Kontrollfluss (`if`, `while`) auf explizite Instruktionsfolgen ist ein zentraler Schritt vom AST zum Zielcode.
- [MUSS] Mapping `if/else` in WASM herausstellen: Bedingung auf Stack -> `if` -> optional `else` -> `end`.
- [MUSS] Mapping `while` in WASM herausstellen: `block` + `loop`, invertierte Bedingung mit `br_if`, Rücksprung mit `br 0`.
- [MUSS] Return-Verhalten dokumentieren: explizites `return` im AST + Fallback-Return im generierten Code.
- [NICE] Kurze Tabelle `AST-Konstrukt -> IR -> WASM` (je 1 Mini-Beispiel für Addition, Vergleich, `if/else`, `while`).
- [NICE] Offener Punkt: Dead Code nach `return` wird noch emittiert (keine DCE/CFG-Bereinigung).
- [STREICHEN] zu viele WASM-Details ohne direkten Bezug zu deinem eigenen Codepfad.

```rust
pub enum IrInstruction{
    I64Const(i64),
    I64Eqz,
    BrIf(u32),
    Br(u32),
    LocalSet(u32),
    LocalGet(u32),
    Call(u32),
    If(BlockType),
    Else,
    Block(BlockType),
    Loop(BlockType),
    Drop,
    Return,
    End,

    // Arithmetik
    I64Add,
    I64Sub,
    I64Mul,
    I64DivS,
    I64Eq,
    I64LtS,
    I64GtS,
    I64ExtendI32S,
    I32Eqz,
}
```

Codebeispiel 15: IR-Instruktionen (Quelle: src/codegen/ir.rs)

Erläuterung:

- `I64Const` entspricht dem Laden einer Konstante im WASM-Stack.
- `LocalGet/LocalSet` stehen für Variablenzugriffe.
- `I64Add` und `I64Eq` sind direkte WASM-Arithmetik/Vergleiche.
- Kontrollfluss wird über `If/Else/Block/Loop/End` abgebildet.

```rust
pub fn emit_function(&mut self, func: &FunctionDecl) {
    let mut r#gen = FuncGen {
        locals: Vec::new(),
        local_map: HashMap::new(),
        instructions: Vec::new(),
        has_return: func.return_type.is_some(),
    };

    // Parameter werden auf lokale Indizes gemappt
    for (i, name) in func.params.iter().enumerate() {
        r#gen.local_map.insert(name.clone(), i as u32);
    }

    // Statements -> IR-Instruktionen
    for stmt in &func.body {
        emit_stmt(stmt, &mut r#gen, &self.func_indices);
    }
}
```

Codebeispiel 16: Funktions-Emission (Quelle: src/codegen/module.rs)

Erläuterung:

- Parameter werden zu lokalen Indizes abgebildet.
- Statements erzeugen eine lineare Folge von IR-Instruktionen.
- Danach folgt die Umwandlung der IR-Instruktionen in echtes WASM.

3.5 Ausführung und Beispiel

- WASM wird lokal ausgeführt, nicht im Browser.
- Runtime: `wasmtime` (lädt Bytecode, instanziert Modul, ruft `main` auf).
- Host-Import `print_i64` wird in Rust bereitgestellt, damit `print(...)` funktioniert.
- Ablauf: Quelltext → Tokens → AST → WASM-Bytes → Wasmtime ausführen.
- `Engine`, `Module`, `Store` und `Instance` sind die zentralen Bausteine der Wasmtime-Ausführung [12].
- Imports müssen beim Instanziieren bereitgestellt werden, sonst schlägt das Laden des Moduls fehl [13].
- [MUSS] Konvention des Projekts explizit machen: Einstieg über exportierte `main`.
- [MUSS] Validierung erwähnen: ungültiges WASM fällt beim Laden/Instanziieren auf.
- [NICE] Grenzen der Laufzeit klar benennen (keine Dateisystem-/Netzwerkzugriffe ohne Imports; nur freigegebene Host-Funktionen).
- [NICE] Messpunkt ergänzen: Compile-Zeit vs. Laufzeitzeit für 1-2 Testprogramme.

```rust
pub fn run_wasm_bytes(bytes: &[u8], args: Vec<i64>) -> Result<Option<i64>,
String> {
    let engine = Engine::default();
    let module = wasmtime::Module::from_binary(&engine, bytes)
        .map_err(|e| format!("module compile error: {}", e))?;

    let mut store = Store::new(&engine, ());

    // Host-Funktion für print(...)
    let print_func = wasmtime::Func::wrap(&mut store, |v: i64| {
        println!("{}", v);
    });

    let instance = Instance::new(&mut store, &module, &[print_func.into()])
        .map_err(|e| format!("instance error: {}", e))?;

    let func = instance
        .get_func(&mut store, "main")
        .ok_or_else(|| "function `main` not found".to_string())?;
    // ...
    func.call(&mut store, &params, &mut results_buf)
        .map_err(|e| format!("runtime error: {}", e))?;
    // ...
}
```

Codebeispiel 17: WASM-Ausführung mit Wasmtime (Quelle: src/runner.rs)

Beispiel (Fakultät):

```
fn fact(n) -> Int {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

fn main(){
    print(fact(10));
    return;
}
```

Codebeispiel 18: Programmbeispiel: Fakultät (Quelle: factorial.eres)

3.6 Eigener Beitrag und verwendete Tools

- Eigenleistung: Sprachsyntax festgelegt (`fn`, `let`, `if/else`, `while`, `return`)
- Eigenleistung: Lexer, Parser, Codegen, Runtime-Anbindung umgesetzt
- Eigenleistung: Testfälle für Lexer, Parser und Ausführung ergänzt
- Verwendete Tools: Rust
- Verwendete Tools: `wasm-encoder`
- Verwendete Tools: `wasmtime`
- Verwendete Tools: Typst
- [NICE] Unterstützung: KI für Strukturierung/Brainstorming

3.7 Testwerkzeuge und CLI-Nutzung

- Token-Ausgabe: `cargo run -- <datei> --print-tokens`
- AST-Ausgabe: `cargo run -- <datei> --print-ast`
- WAT-Ausgabe: `cargo run -- <datei> --print-wat`
- Artefakt: Token-Stream (Lexer)
- Artefakt: AST (Parser)
- Artefakt: WAT (Codegen)
- [MUSS] Fehlerfall demonstrieren (z.B. unerwartetes Token) inkl. Position/Context der Fehlermeldung.
- [MUSS] Testmatrix (kurz, tabellarisch): arithmetische Ausdrücke, Vergleich + `if/else`, Schleife, Rekursion (`fact`), Grenzfall (leere Parameterliste/fehlendes Semikolon).
- [NICE] Gegenüberstellung „erwartete Ausgabe vs. tatsächliche Ausgabe“ für 2-3 Programme.
- [STREICHEN] zu viele CLI-Varianten ohne Erkenntnisgewinn (auf Kernbefehle begrenzen).

4 Fazit

- Zusammenfassung der zentralen Ergebnisse
- Persönliche Stellungnahme zur Leitfrage
- Selbstreflexion: Herausforderungen und Verbesserungen
- Zusammenfassung: WASM reduziert Backend-Komplexität
- Persönliche Stellungnahme: Ziel der Arbeit erreicht / Leitfrage beantwortet
- Selbstreflexion: größte Herausforderungen (Parserlogik, Codegen, Tooling)
- Selbstreflexion: nächste Iteration (Fehlerdiagnostik, Typen, Sprachumfang)
- [MUSS] Leitfrage final gewichten: WASM senkt Einstiegshürde im Backend, Gesamtkomplexität bleibt mittel-hoch wegen Frontend + Semantik.
- [MUSS] Konkrete Antwortformel für den Schlusssatz: „erleichtert deutlich, ersetzt aber kein Compiler-Grundwissen“.
- [NICE] Ausblick (nächste Iteration): Typchecker, bessere Fehlermeldungen (Zeile/Spalte + Recovery), kleine Optimierungen (constant folding, dead code).
- [STREICHEN] Wiederholung der Einleitungsaussagen ohne neue Bewertung.

5 Quellen

[1] WebAssembly Community Group, „WebAssembly Core Specification“. [Online]. Verfügbar unter: https://webassembly.github.io/spec/

[2] IBM, „Was ist ein Compiler?“. [Online]. Verfügbar unter: https://www.ibm.com/de-de/think/topics/compiler

[3] GNU Project, „Overall Options (Using the GNU Compiler Collection)“. [Online]. Verfügbar unter: https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc/Overall-Options.html

[4] Rust Compiler Development Guide, „Overview of the compiler“. [Online]. Verfügbar unter: https://rustc-dev-guide.rust-lang.org/overview.html

[5] Rust Compiler Development Guide, „Lexing and parsing“. [Online]. Verfügbar unter: https://rustc-dev-guide.rust-lang.org/the-parser.html

[6] MDN Web Docs, „WebAssembly concepts“. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts

[7] MDN Web Docs, „Understanding WebAssembly text format“. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format

[8] W3C, „WebAssembly Core Specification (W3C Working Draft)“. [Online]. Verfügbar unter: https://www.w3.org/TR/2024/WD-wasm-core-2-20240217/

[9] Robert Nystrom, „Scanning (Chapter 4) - Crafting Interpreters“. [Online]. Verfügbar unter: https://craftinginterpreters.com/scanning.html

[10] Robert Nystrom, „Parsing Expressions (Chapter 6) - Crafting Interpreters“. [Online]. Verfügbar unter: https://craftinginterpreters.com/parsing-expressions.html

[11] LLVM Project, „LLVM Language Reference Manual“. [Online]. Verfügbar unter: https://llvm.org/docs/LangRef.html

[12] Bytecode Alliance, „wasmtime crate documentation“. [Online]. Verfügbar unter: https://docs.rs/wasmtime/latest/wasmtime/

[13] Bytecode Alliance, „Hello, world! - Wasmtime“. [Online]. Verfügbar unter: https://docs.wasmtime.dev/examples-hello-world.html

6 Anhang: Quellcode

Anhang-Inhaltsverzeichnis:

Quellcode-Übersicht

6.0.1 factorial.eres

[

```
// factorial example
fn fact(n) -> Int {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

fn main(){
    print(fact(10));
    return;
}
```

]

6.0.2 src/ast.rs

[

```
#[derive(Debug)]
pub enum Expr {
    Int(i64),
    Local(String),
    Binary {
        op: BinOp,
        left: Box<Expr>,
        right: Box<Expr>,
    },
    Call {
        name: String,
        args: Vec<Expr>,
    },
}

#[derive(Debug)]
pub enum BinOp {
    Add,
    Sub,
    Mul,
    Div,
    Eq,
    NotEq,
    Lt,
    Le,
    Gt,
    Ge,
}

#[derive(Debug)]
pub enum Stmt {
    Let {
        name: String,
```

```
        value: Expr,
    },
    Return(Option<Expr>),
    Expr(Expr),
    If {
        cond: Expr,
        then_block: Vec<Stmt>,
        else_block: Vec<Stmt>,
    },
    While {
        cond: Expr,
        body: Vec<Stmt>,
    },
}

#[derive(Debug)]
pub struct FunctionDecl {
    pub name: String,
    pub params: Vec<String>,
    pub body: Vec<Stmt>,
    pub return_type: Option<Type>,
}

#[derive(Debug)]
pub enum Type {
    Int,
}

#[derive(Debug)]
pub struct Program {
    pub functions: Vec<FunctionDecl>,
}
```

]

6.0.3 src/lexer.rs

[

```
use std::str::Chars;

use crate::token::{Span, Token, TokenKind};

pub struct Lexer<'a> {
    chars: Chars<'a>,
    pos: usize,
}

impl<'a> Lexer<'a> {
    fn peek(&self) -> Option<char> {
        self.chars.clone().next()
    }

    fn bump(&mut self) -> Option<char> {
        let c = self.chars.next()?;
        self.pos += c.len_utf8();
        Some(c)
```

```
}

fn span_from(&self, start: usize) -> Span {
    Span {
        start,
        end: self.pos,
    }
}

pub fn next_token(&mut self) -> Result<Token, LexError> {
    self.skip_whitespace();

    let start = self.pos;

    let ch = match self.bump() {
        Some(c) => c,
        None => {
            return Ok(Token {
                kind: TokenKind::EOF,
                span: Span { start, end: start },
            });
        }
    };

    let kind = match ch {
        '(' => TokenKind::LParen,
        ')' => TokenKind::RParen,
        '{' => TokenKind::LBrace,
        '}' => TokenKind::RBrace,
        ';' => TokenKind::Semicolon,
        '+' => TokenKind::Plus,
        ':' => TokenKind::Colon,
        ',' => TokenKind::Comma,
        '>' => {
            if let Some('=') = self.peek() {
                self.bump();
                TokenKind::GreaterEqual
            } else {
                TokenKind::Greater
            }
        }
        '<' => {
            if let Some('=') = self.peek() {
                self.bump();
                TokenKind::LessEqual
            } else {
                TokenKind::Less
            }
        }
        '-' => {
            if let Some('>') = self.peek() {
                self.bump();
                TokenKind::Arrow
            } else {
                TokenKind::Minus
            }
```

```
            }
            '*' => TokenKind::Star,
            '/' => {
                if let Some('/') = self.peek() {
                    // Single-line comment
                    while let Some(c) = self.peek() {
                        if c == '\n' {
                            break;
                        }
                        self.bump();
                    }
                    return self.next_token();
                } else {
                    TokenKind::Slash
                }
            }
            '%' => TokenKind::Percentage,
            '=' => {
                if let Some('=') = self.peek() {
                    self.bump();
                    TokenKind::EqualEqual
                } else {
                    TokenKind::Equal
                }
            }
            '!' => {
                if let Some('=') = self.peek() {
                    self.bump();
                    TokenKind::NotEqual
                } else {
                    return Err(LexError::UnexpectedChar {
                        ch: '!',
                        span: self.span_from(start),
                    });
                }
            }
            c if c.is_ascii_digit() => self.lex_number(c)?,
            c if Lexer::is_ident_start(c) => self.lex_ident(c),

            _ => {
                return Err(LexError::UnexpectedChar {
                    ch,
                    span: self.span_from(start),
                });
            }
        };

        Ok(Token {
            kind,
            span: self.span_from(start),
        })
    }

    pub fn lex_number(&mut self, first_digit: char) -> Result<TokenKind, LexError>
{
        let start = self.pos - first_digit.len_utf8();
```

```
    let mut number_str = first_digit.to_string();

    while let Some(c) = self.peek() {
        if c.is_ascii_digit() {
            number_str.push(c);
            self.bump();
        } else {
            break;
        }
    }

    match number_str.parse::<i64>() {
        Ok(value) => Ok(TokenKind::Int(value)),
        Err(_) => Err(LexError::InvalidNumber {
            span: self.span_from(start),
        }),
    }
}

pub fn is_ident_start(c: char) -> bool {
    c.is_ascii_alphabetic() || c == '_'
}

pub fn lex_ident(&mut self, first_char: char) -> TokenKind {
    let mut ident_str = first_char.to_string();

    while let Some(c) = self.peek() {
        if c.is_ascii_alphanumeric() || c == '_' {
            ident_str.push(c);
            self.bump();
        } else {
            break;
        }
    }

    match ident_str.as_str() {
        "let" => TokenKind::Let,
        "fn" => TokenKind::Fn,
        "if" => TokenKind::If,
        "else" => TokenKind::Else,
        "while" => TokenKind::While,
        "return" => TokenKind::Return,
        "Int" => TokenKind::IntType,
        _ => TokenKind::Ident(ident_str),
    }
}

fn skip_whitespace(&mut self) {
    while let Some(c) = self.peek() {
        if c.is_whitespace() {
            self.bump();
        } else {
            break;
        }
    }
}
```

```rust
    pub fn new(src: &'a str) -> Self {
        Lexer {
            chars: src.chars(),
            pos: 0,
        }
    }
}

pub fn lex_file(src: &str) -> Result<Vec<Token>, LexError> {
    let mut lexer = Lexer::new(src);
    let mut tokens = Vec::new();

    loop {
        let token = lexer.next_token()?;
        if token.kind == TokenKind::EOF {
            tokens.push(token);
            break;
        }
        tokens.push(token);
    }

    Ok(tokens)
}

#[derive(Debug)]
pub enum LexError {
    UnexpectedChar { ch: char, span: Span },
    InvalidNumber { span: Span },
}

pub fn report_lex_error(_src: &str, error: LexError) {
    match error {
        LexError::UnexpectedChar { ch, span } => {
            eprintln!(
                "LexError: Unexpected character '{}' at {}:{}",
                ch, span.start, span.end
            );
        }
        LexError::InvalidNumber { span } => {
            eprintln!("LexError: Invalid number at {}:{}", span.start, span.end);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn lex_simple_tokens() {
        let src = "( ) { } ; + - * / % , : -> <= < >= > == != ->";
        // include arrow twice intentionally
        let tokens = lex_file(src).expect("lexing failed");
        // ensure EOF is last
        assert_eq!(tokens.last().unwrap().kind, TokenKind::EOF);
```

```
        // Check that some known tokens appear in order (spot check)
        let kinds: Vec<_> = tokens.iter().map(|t| &t.kind).collect();
        assert!(kinds.contains(&&TokenKind::LParen));
        assert!(kinds.contains(&&TokenKind::LessEqual));
        assert!(kinds.contains(&&TokenKind::GreaterEqual));
        assert!(kinds.contains(&&TokenKind::EqualEqual));
        assert!(kinds.contains(&&TokenKind::NotEqual));
    }
}
```

]

6.0.4 src/main.rs

[

```
use clap::Parser;

use crate::codegen::module::ModuleGen;

mod ast;
mod codegen;
mod lexer;
mod parser;
mod runner;
mod token;

#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Input source file
    input: String,

    /// Print tokens produced by the lexer
    #[arg(long, default_value_t = false)]
    print_tokens: bool,

    /// Print the parsed AST
    #[arg(long, default_value_t = false)]
    print_ast: bool,

    /// Print generated WAT before running
    #[arg(long, default_value_t = false)]
    print_wat: bool,
}

fn main() {
    let args = Args::parse();

    let src = std::fs::read_to_string(&args.input).expect("Failed to read input
file");

    let bytes = match lexer::lex_file(&src) {
        Ok(tokens) => {
            if args.print_tokens {
                println!("Tokens:\n{:#?}", tokens);
            }
```

```
            let mut parser = parser::Parser::new(&tokens);
            let ast = parser.parse_program().unwrap();

            if args.print_ast {
                println!("AST:\n{:#?}", ast);
            }

            let mut module_gen = ModuleGen::new();
            module_gen = module_gen.init_with_host_functions();

            // Declare first, then emit, so functions can reference each other.
            for func in &ast.functions {
                module_gen.declare_function(func);
            }
            for func in &ast.functions {
                module_gen.emit_function(func);
            }

            let bytes = module_gen.finish();

            // Determine the arity of main to prepare call arguments.
            let main_param_count = ast
                .functions
                .iter()
                .find(|f| f.name == "main")
                .map(|f| f.params.len())
                .unwrap_or(0);

            Some((bytes, args.print_wat, main_param_count))
        }
        Err(e) => {
            lexer::report_lex_error(&src, e);
            None
        }
    };

    if let Some((bytes, print_wat, param_count)) = bytes {
        let wat = wasmprinter::print_bytes(&bytes).unwrap();
        if print_wat {
            println!("Generated WAT:\n{}", wat);
        }

        // Prepare zero-initialized i64 args matching the function's param count.
        let args: Vec<i64> = vec![0; param_count];

        match runner::run_wasm_bytes(&bytes, args) {
            Ok(Some(result)) => println!("result of main function: {}", result),
            Ok(None) => println!("main returned no value"),
            Err(e) => eprintln!("Execution error: {}", e),
        }
    }
}


]
```

6.0.5 src/parser.rs

[

```
use crate::ast::*;
use crate::token::*;

#[derive(Debug)]
pub struct Parser<'a> {
    tokens: &'a [Token],
    pos: usize,
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::lexer::lex_file;

    #[test]
    fn parse_paren_comparison() {
        let src = "fn main(x, y) -> Int { if (x < y) { return 1; } else { return
0; } }";
        let tokens = lex_file(src).expect("lex");
        let mut p = Parser::new(&tokens);
        let program = p.parse_program().expect("parse");

        // one function
        assert_eq!(program.functions.len(), 1);
        let f = &program.functions[0];
        assert_eq!(f.body.len(), 1);

        match &f.body[0] {
            Stmt::If {
                cond,
                then_block,
                else_block,
            } => {
                match cond {
                    Expr::Binary { op, left, right } => {
                        match op {
                            BinOp::Lt => (),
                            _ => panic!("expected Lt op"),
                        }

                        match (&**left, &**right) {
                            (Expr::Local(l), Expr::Local(r)) => {
                                assert_eq!(l, "x");
                                assert_eq!(r, "y");
                            }
                            _ => panic!("expected locals in comparison"),
                        }
                    }
                    _ => panic!("expected binary cond"),
                }

                assert_eq!(then_block.len(), 1);
                assert_eq!(else_block.len(), 1);
```

```
        }
        _ => panic!("expected if statement"),
    }
}

#[test]
fn parse_mixed_precedence() {
    let src = "fn main(x, y) -> Int { let z = x + 1 < y; }";
    let tokens = lex_file(src).expect("lex");
    let mut p = Parser::new(&tokens);
    let program = p.parse_program().expect("parse");
    let f = &program.functions[0];

    match &f.body[0] {
        Stmt::Let { name, value } => {
            assert_eq!(name, "z");
            match value {
                Expr::Binary { op, left, right } => {
                    match op {
                        BinOp::Lt => (),
                        _ => panic!("expected Lt op"),
                    }
                    match &**left {
                        Expr::Binary {
                            op: lop,
                            left: lleft,
                            right: _lright,
                        } => {
                            match lop {
                                BinOp::Add => (),
                                _ => panic!("expected Add as left op"),
                            }
                            match &**lleft {
                                Expr::Local(n) => assert_eq!(n, "x"),
                                _ => panic!("expected local x"),
                            }
                        }
                        _ => panic!("expected binary left side"),
                    }
                    match &**right {
                        Expr::Local(n) => assert_eq!(n, "y"),
                        _ => panic!("expected local y"),
                    }
                }
                _ => panic!("expected binary value"),
            }
        }
        _ => panic!("expected let statement"),
    }
}

#[test]
fn parse_parenthesized_complex() {
    let src = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); }";
    let tokens = lex_file(src).expect("lex");
    let mut p = Parser::new(&tokens);
```

```
        let program = p.parse_program().expect("parse");
        let f = &program.functions[0];

        match &f.body[0] {
            Stmt::Let { name, value } => {
                assert_eq!(name, "r");
                match value {
                    Expr::Binary {
                        op,
                        left: _left,
                        right: _right,
                    } => match op {
                        BinOp::Le => (),
                        _ => panic!("expected Le op"),
                    },
                    _ => panic!("expected binary value"),
                }
            }
            _ => panic!("expected let statement"),
        }
    }
}

impl<'a> Parser<'a> {
    pub fn new(tokens: &'a [Token]) -> Parser<'a> {
        Parser { tokens, pos: 0 }
    }

    fn peek(&self) -> &Token {
        self.tokens.get(self.pos).unwrap()
    }

    fn bump(&mut self) -> Token {
        let tok = self.peek().clone();
        self.pos += 1;
        tok
    }

    fn expect(&mut self, kind: TokenKind) -> Result<Token, ParseError> {
        let tok = self.bump();
        if tok.kind == kind {
            Ok(tok)
        } else {
            Err(ParseError::UnexpectedToken {
                expected: kind.name(),
                found: tok,
            })
        }
    }

    pub fn parse_program(&mut self) -> Result<Program, ParseError> {
        let mut functions = Vec::new();

        while self.peek().kind != TokenKind::EOF {
            let func = self.parse_function()?;
            functions.push(func);
```

```
    }
    Ok(Program { functions })
}

fn parse_function(&mut self) -> Result<FunctionDecl, ParseError> {
    self.expect(TokenKind::Fn)?;

    let name = match self.bump().kind.clone() {
        TokenKind::Ident(s) => s,
        tok => {
            return Err(ParseError::UnexpectedToken {
                expected: "identifier".to_string(),
                found: Token {
                    kind: tok,
                    span: self.peek().span.clone(),
                },
            });
        }
    };

    self.expect(TokenKind::LParen)?;

    let mut params = Vec::new();
    if self.peek().kind != TokenKind::RParen {
        loop {
            match self.bump().kind.clone() {
                TokenKind::Ident(s) => params.push(s),
                _ => {
                    return Err(ParseError::UnexpectedToken {
                        expected: "parameter name".to_string(),
                        found: self.peek().clone(),
                    });
                }
            }

            if self.peek().kind == TokenKind::Comma {
                self.bump();
            } else {
                break;
            }
        }
    }

    self.expect(TokenKind::RParen)?;
    // Optional return type (`-> Int`).
    let return_type = if self.peek().kind == TokenKind::Arrow {
        self.bump();
        // Currently only Int is supported.
        self.expect(TokenKind::IntType)?;
        Some(crate::ast::Type::Int)
    } else {
        None
    };
    let body = self.parse_block()?;

    Ok(FunctionDecl {
```

```
            name,
            params,
            body,
            return_type,
        })
    }

    fn parse_block(&mut self) -> Result<Vec<Stmt>, ParseError> {
        self.expect(TokenKind::LBrace)?;
        let mut stmts = Vec::new();

        while self.peek().kind != TokenKind::RBrace {
            let stmt = self.parse_stmt()?;
            stmts.push(stmt);
        }

        self.expect(TokenKind::RBrace)?;
        Ok(stmts)
    }

    fn parse_stmt(&mut self) -> Result<Stmt, ParseError> {
        match self.peek().kind {
            TokenKind::Let => self.parse_let(),
            TokenKind::Return => self.parse_return(),
            TokenKind::If => self.parse_if(),
            TokenKind::While => self.parse_while(),
            _ => self.parse_expr_stmt(),
        }
    }

    fn parse_let(&mut self) -> Result<Stmt, ParseError> {
        self.expect(TokenKind::Let)?;

        let name = match self.bump().kind.clone() {
            TokenKind::Ident(s) => s,
            _ => {
                return Err(ParseError::UnexpectedToken {
                    expected: "identifier".to_string(),
                    found: self.peek().clone(),
                });
            }
        };

        self.expect(TokenKind::Equal)?;
        let value = self.parse_expr()?;
        self.expect(TokenKind::Semicolon)?;

        Ok(Stmt::Let { name, value })
    }

    fn parse_return(&mut self) -> Result<Stmt, ParseError> {
        self.expect(TokenKind::Return)?;

        // Allow `return;` with no expression. If present, parse an expression.
        if self.peek().kind == TokenKind::Semicolon {
            self.bump();
```

```
            Ok(Stmt::Return(None))
        } else {
            let expr = self.parse_expr()?;
            self.expect(TokenKind::Semicolon)?;
            Ok(Stmt::Return(Some(expr)))
        }
    }

    fn parse_if(&mut self) -> Result<Stmt, ParseError> {
        self.expect(TokenKind::If)?;

        let cond = self.parse_expr()?;

        let then_block = self.parse_block()?;

        let else_block = if self.peek().kind == TokenKind::Else {
            self.bump();
            self.parse_block()?
        } else {
            Vec::new()
        };

        Ok(Stmt::If {
            cond,
            then_block,
            else_block,
        })
    }

    fn parse_while(&mut self) -> Result<Stmt, ParseError> {
        self.expect(TokenKind::While)?;
        let cond = self.parse_expr()?;
        let body = self.parse_block()?;

        Ok(Stmt::While { cond, body })
    }

    fn parse_expr_stmt(&mut self) -> Result<Stmt, ParseError> {
        let expr = self.parse_expr()?;
        self.expect(TokenKind::Semicolon)?;
        Ok(Stmt::Expr(expr))
    }

    fn parse_expr(&mut self) -> Result<Expr, ParseError> {
        self.parse_binary_expr(0)
    }

    // Precedence-climbing / Pratt-style binary expression parser
    // min_prec: minimum precedence to accept in this call
    fn parse_binary_expr(&mut self, min_prec: u8) -> Result<Expr, ParseError> {
        // 1. parse left-hand side primary
        let mut lhs = self.parse_primary()?;

        loop {
            // 2. check if next token is a binary operator and get its precedence
            let (op, prec) = match self.token_to_binop_prec(&self.peek().kind) {
```

```
                Some(x) => x,
                None => break,
            };

            if prec < min_prec {
                break;
            }

            // 3. consume operator
            self.bump();

            // 4. parse RHS with higher precedence (left-associative operators use
prec+1)
            let next_min = prec + 1;
            let rhs = self.parse_binary_expr(next_min)?;

            // 5. build binary node and continue
            lhs = Expr::Binary {
                op,
                left: Box::new(lhs),
                right: Box::new(rhs),
            };
        }

        Ok(lhs)
    }

    // Map a token kind to (BinOp, precedence). Higher number = higher precedence.
    // Precedence levels (high -> low): * / = 4, + - = 3, < > <= >= = 2, == != = 1
    fn token_to_binop_prec(&self, kind: &TokenKind) -> Option<(BinOp, u8)> {
        match kind {
            TokenKind::Star => Some((BinOp::Mul, 4)),
            TokenKind::Slash => Some((BinOp::Div, 4)),
            TokenKind::Plus => Some((BinOp::Add, 3)),
            TokenKind::Minus => Some((BinOp::Sub, 3)),
            TokenKind::Less => Some((BinOp::Lt, 2)),
            TokenKind::LessEqual => Some((BinOp::Le, 2)),
            TokenKind::Greater => Some((BinOp::Gt, 2)),
            TokenKind::GreaterEqual => Some((BinOp::Ge, 2)),
            TokenKind::EqualEqual => Some((BinOp::Eq, 1)),
            TokenKind::NotEqual => Some((BinOp::NotEq, 1)),
            _ => None,
        }
    }

    fn parse_primary(&mut self) -> Result<Expr, ParseError> {
        let tok = self.bump().clone();

        match tok.kind {
            TokenKind::Int(value) => Ok(Expr::Int(value)),

            TokenKind::Ident(name) => {
                if self.peek().kind == TokenKind::LParen {
                    self.bump(); // '('
                    let mut args = Vec::new();
```

```
                    if self.peek().kind != TokenKind::RParen {
                        loop {
                            args.push(self.parse_expr()?);

                            if self.peek().kind == TokenKind::Comma {
                                self.bump();
                            } else {
                                break;
                            }
                        }
                    }

                    self.expect(TokenKind::RParen)?;
                    Ok(Expr::Call { name, args })
                } else {
                    Ok(Expr::Local(name))
                }
            }

            TokenKind::LParen => {
                let expr = self.parse_expr()?;
                self.expect(TokenKind::RParen)?;
                Ok(expr)
            }
            _ => Err(ParseError::ExpectedExpression { span: tok.span }),
        }
    }
}

#[allow(dead_code)]
#[derive(Debug)]
pub enum ParseError {
    UnexpectedToken { expected: String, found: Token },
    ExpectedExpression { span: Span },
}
```

]

6.0.6 src/runner.rs

[

```
use std::sync::{Arc, Mutex};
use wasmtime::{Engine, Instance, Store, Val};

use crate::codegen::module::ModuleGen;
use crate::lexer::lex_file;
use crate::parser::Parser;

/// Compile source to wasm bytes.
pub fn compile_bytes_from_src(src: &str) -> Result<Vec<u8>, String> {
    let tokens = lex_file(src).map_err(|e| format!("lex error: {:?}", e))?;
    let mut parser = Parser::new(&tokens);
    let program = parser
        .parse_program()
        .map_err(|e| format!("parse error: {:?}", e))?;
```

```
    // Register host imports (print), then declare and emit all functions.
    let mut module_gen = ModuleGen::new().init_with_host_functions();
    for func in &program.functions {
        module_gen.declare_function(func);
    }
    for func in &program.functions {
        module_gen.emit_function(func);
    }
    let bytes = module_gen.finish();
    Ok(bytes)
}

/// Run wasm bytes calling `main` with the provided i64 arguments.
/// Returns Ok(Some(i64)) if the function returns a single i64, Ok(None) if
/// the function has no return, or Err on failure.
pub fn run_wasm_bytes(bytes: &[u8], args: Vec<i64>) -> Result<Option<i64>, String>
{
    let engine = Engine::default();
    let module = wasmtime::Module::from_binary(&engine, bytes)
        .map_err(|e| format!("module compile error: {}", e))?;

    let mut store = Store::new(&engine, ());

    // create host print function
    let print_func = wasmtime::Func::wrap(&mut store, |v: i64| {
        println!("{}", v);
    });

    let instance = Instance::new(&mut store, &module, &[print_func.into()])
        .map_err(|e| format!("instance error: {}", e))?;

    let func = instance
        .get_func(&mut store, "main")
        .ok_or_else(|| "function `main` not found".to_string())?;

    // Prepare Val parameters
    let params: Vec<Val> = args.into_iter().map(Val::I64).collect();

    // Inspect function type to determine result count
    let ty = func.ty(&store);
    let results = ty.results().len();

    let mut results_buf: Vec<Val> = vec![Val::I64(0); results];

    func.call(&mut store, &params, &mut results_buf)
        .map_err(|e| format!("runtime error: {}", e))?;

    if results == 1 {
        if let Val::I64(v) = results_buf[0] {
            Ok(Some(v))
        } else {
            Err("unexpected return value type".to_string())
        }
    } else {
        Ok(None)
    }
```

```
}

/// Convenience helper: compile source and run it.
pub fn run_source(src: &str, args: Vec<i64>) -> Result<Option<i64>, String> {
    let bytes = compile_bytes_from_src(src)?;
    run_wasm_bytes(&bytes, args)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn run_if_gt_sample() {
        let src = "fn main(x, y) -> Int { if (x > 5) { return 1; } else { return
0; } }";
        let res = run_source(src, vec![64, 8]).expect("run failed");
        assert_eq!(res, Some(1));
    }

    #[test]
    fn run_add_compare() {
        let src = "fn main(a, b) -> Int { let z = a + 1 < b; return z; }";
        // a=1, b=3 -> 1+1 < 3 -> true -> 1
        let res = run_source(src, vec![1, 3]).expect("run failed");
        assert_eq!(res, Some(1));
    }

    #[test]
    fn run_parenthesized_complex() {
        let src = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); return
r; }";
        // Wrong arity should return an error.
        let _res =
            run_source(src, vec![1, 1]).expect_err("expected error because missing
c,d args");

        let src2 = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); return
r; }";
        let bytes = compile_bytes_from_src(src2).expect("compile failed");
        let out =
            run_wasm_bytes(&bytes, vec![2, 1]).expect_err("expected error due to
wrong arg count");
        let _ = out;
    }

    #[test]
    fn print_statement_outputs() {
        // inline source instead of reading a file
        let src = "fn main(){ print(3); return; }";

        let bytes = compile_bytes_from_src(src).expect("compile failed");

        // instantiate module with a host `print_i64` that records values
        let engine = Engine::default();
        let module = wasmtime::Module::from_binary(&engine, &bytes).expect("module
compile failed");
```

```
        let printed: Arc<Mutex<Vec<i64>>> = Arc::new(Mutex::new(Vec::new()));

        let mut store = Store::new(&engine, ());
        let captured = printed.clone();
        let print_func = wasmtime::Func::wrap(&mut store, move |v: i64| {
            captured.lock().unwrap().push(v);
        });

        let instance = Instance::new(&mut store, &module, &[print_func.into()])
            .expect("instance creation failed");

        let func = instance
            .get_func(&mut store, "main")
            .expect("main not found");
        let params: Vec<Val> = Vec::new();
        let mut results_buf: Vec<Val> = Vec::new();

        func.call(&mut store, &params, &mut results_buf)
            .expect("call failed");

        let got = printed.lock().unwrap().clone();
        assert_eq!(got, vec![3]);
    }
}
```

]

6.0.7 src/token.rs

[

```
#[derive(Debug, Clone, PartialEq)]
pub enum TokenKind {
    // Keywords
    Let,
    Fn,
    If,
    Else,
    While,
    Return,

    // Identifiers + literals
    Ident(String),
    Int(i64),

    // Operators
    Plus,
    Minus,
    Star,
    Slash,
    Percentage,
    Equal,

    // Comparison operators
    EqualEqual,
    NotEqual,
```

```
    Less,
    LessEqual,
    Greater,
    GreaterEqual,

    // Delimiters
    LParen,
    RParen,
    LBrace,
    RBrace,
    Semicolon,
    Comma,

    // Type hints
    Colon,
    Arrow,
    IntType,

    EOF,
}

impl TokenKind {
    pub fn name(&self) -> String {
        match self {
            TokenKind::Ident(_) => "identifier".to_string(),
            TokenKind::Int(_) => "integer literal".to_string(),
            _ => format!("{:?}", self),
        }
    }
}

#[derive(Debug, Clone)]
pub struct Token {
    pub kind: TokenKind,
    pub span: Span,
}

#[derive(Debug, Clone)]
pub struct Span {
    pub start: usize,
    pub end: usize,
}
```

]

6.0.8 src/codegen/expr.rs

[

```
use std::collections::HashMap;

use crate::{
    ast::{BinOp, Expr},
    codegen::ir::IrInstruction,
    codegen::module::FuncGen,
};
```

```rust
pub fn emit_expr(expr: &Expr, r#gen: &mut FuncGen, funcs: &HashMap<String, (u32,
bool)>) -> bool {
    match expr {
        Expr::Int(v) => {
            r#gen.instructions.push(IrInstruction::I64Const(*v));
            true
        }
        Expr::Local(name) => {
            let idx = r#gen.local_map[name];
            r#gen.instructions.push(IrInstruction::LocalGet(idx));
            true
        }
        Expr::Binary { op, left, right } => {
            let _ = emit_expr(left, r#gen, funcs);
            let _ = emit_expr(right, r#gen, funcs);

            match op {
                BinOp::Add => r#gen.instructions.push(IrInstruction::I64Add),
                BinOp::Sub => r#gen.instructions.push(IrInstruction::I64Sub),
                BinOp::Mul => r#gen.instructions.push(IrInstruction::I64Mul),
                BinOp::Div => r#gen.instructions.push(IrInstruction::I64DivS),
                BinOp::Eq => {
                    r#gen.instructions.push(IrInstruction::I64Eq);
                    // Keep expression results as i64 across the compiler.
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                BinOp::Lt => {
                    r#gen.instructions.push(IrInstruction::I64LtS);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                BinOp::Gt => {
                    r#gen.instructions.push(IrInstruction::I64GtS);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a != b  =>  !(a == b)
                BinOp::NotEq => {
                    r#gen.instructions.push(IrInstruction::I64Eq);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a <= b  =>  !(a > b)
                BinOp::Le => {
                    r#gen.instructions.push(IrInstruction::I64GtS);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a >= b  =>  !(a < b)
                BinOp::Ge => {
                    r#gen.instructions.push(IrInstruction::I64LtS);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
            }

            true
        }
```

```
        Expr::Call { name, args } => {
            for arg in args {
                let _ = emit_expr(arg, r#gen, funcs);
            }
            let (idx, has_ret) = funcs[name];
            r#gen.instructions.push(IrInstruction::Call(idx));
            has_ret
        }
    }
}


]
```

6.0.9 src/codegen/ir.rs

[

```
use wasm_encoder::{BlockType, Instruction};

pub enum IrInstruction {
    I64Const(i64),
    I64Eqz,
    BrIf(u32),
    Br(u32),
    LocalSet(u32),
    LocalGet(u32),
    Call(u32),
    If(BlockType),
    Else,
    Block(BlockType),
    Loop(BlockType),
    Drop,
    Return,
    End,

    // Arithmetic
    I64Add,
    I64Sub,
    I64Mul,
    I64DivS,
    I64Eq,
    I64LtS,
    I64GtS,
    I64ExtendI32S,
    I32Eqz,
}

impl IrInstruction {
    pub fn to_wasm(&self) -> Instruction<'_> {
        match self {
            IrInstruction::I64Const(v) => Instruction::I64Const(*v),
            IrInstruction::I64Eqz => Instruction::I64Eqz,
            IrInstruction::BrIf(idx) => Instruction::BrIf(*idx),
            IrInstruction::Br(idx) => Instruction::Br(*idx),
            IrInstruction::LocalSet(idx) => Instruction::LocalSet(*idx),
            IrInstruction::LocalGet(idx) => Instruction::LocalGet(*idx),
            IrInstruction::Call(idx) => Instruction::Call(*idx),
```

```
            IrInstruction::If(block_type) => Instruction::If(*block_type),
            IrInstruction::Else => Instruction::Else,
            IrInstruction::Block(block_type) => Instruction::Block(*block_type),
            IrInstruction::Loop(block_type) => Instruction::Loop(*block_type),
            IrInstruction::Drop => Instruction::Drop,
            IrInstruction::Return => Instruction::Return,
            IrInstruction::End => Instruction::End,
            IrInstruction::I64Add => Instruction::I64Add,
            IrInstruction::I64Sub => Instruction::I64Sub,
            IrInstruction::I64Mul => Instruction::I64Mul,
            IrInstruction::I64DivS => Instruction::I64DivS,
            IrInstruction::I64Eq => Instruction::I64Eq,
            IrInstruction::I64LtS => Instruction::I64LtS,
            IrInstruction::I64GtS => Instruction::I64GtS,
            IrInstruction::I64ExtendI32S => Instruction::I64ExtendI32S,
            IrInstruction::I32Eqz => Instruction::I32Eqz,
        }
    }
}
```

]

6.0.10 src/codegen/mod.rs

[

```
mod expr;
mod ir;
pub mod module;
mod stmt;
```

]

6.0.11 src/codegen/module.rs

[

```
use std::collections::HashMap;

use crate::codegen::ir::IrInstruction;
use wasm_encoder::*;

use crate::{ast::FunctionDecl, codegen::stmt::emit_stmt};

pub struct ModuleGen {
    module: Module,
    types: TypeSection,
    imports: ImportSection,
    functions: FunctionSection,
    codes: CodeSection,
    exports: ExportSection,

    func_indices: HashMap<String, (u32, bool)>,
    next_type_index: u32,
    next_func_index: u32,
}
```

```rust
impl ModuleGen {
    pub fn new() -> Self {
        Self {
            module: Module::new(),
            types: TypeSection::new(),
            imports: ImportSection::new(),
            functions: FunctionSection::new(),
            codes: CodeSection::new(),
            exports: ExportSection::new(),
            func_indices: HashMap::new(),
            next_type_index: 0,
            next_func_index: 0,
        }
    }

    pub fn init_with_host_functions(mut self) -> Self {
        // Register host imports (currently only print).
        self.add_print_import();
        self
    }

    fn add_print_import(&mut self) {
        // (i64) -> ()
        let type_index = self.next_type_index;
        self.next_type_index += 1;
        self.types.ty().function(vec![ValType::I64], Vec::new());

        // Import as env.print_i64.
        self.imports
            .import("env", "print_i64", EntityType::Function(type_index));

        // Expose this import as `print` in the compiler function table.
        let idx = self.next_func_index;
        self.next_func_index += 1;
        self.func_indices.insert("print".to_string(), (idx, false));
    }

    pub fn finish(mut self) -> Vec<u8> {
        self.module.section(&self.types);
        self.module.section(&self.imports);
        self.module.section(&self.functions);
        self.module.section(&self.exports);
        self.module.section(&self.codes);
        self.module.finish()
    }

    pub fn declare_function(&mut self, func: &FunctionDecl) {
        let type_index = self.next_type_index;
        self.next_type_index += 1;

        let params = vec![ValType::I64; func.params.len()];
        let results = if func.return_type.is_some() {
            vec![ValType::I64]
        } else {
            Vec::new()
        };
```

```
        self.types.ty().function(params, results);

        let idx = self.func_indices.len() as u32;
        self.func_indices
            .insert(func.name.clone(), (idx, func.return_type.is_some()));

        self.functions.function(type_index);
        self.exports.export(&func.name, ExportKind::Func, idx);
    }

    pub fn emit_function(&mut self, func: &FunctionDecl) {
        let mut r#gen = FuncGen {
            locals: Vec::new(),
            local_map: HashMap::new(),
            instructions: Vec::new(),
            has_return: func.return_type.is_some(),
        };

        for (i, name) in func.params.iter().enumerate() {
            r#gen.local_map.insert(name.clone(), i as u32);
        }

        for stmt in &func.body {
            emit_stmt(stmt, &mut r#gen, &self.func_indices);
        }

        // If the function has a return type but no explicit return was emitted,
        // provide a default 0 return so the function type matches. If no return
        // type is declared, don't emit a return value.
        if func.return_type.is_some() {
            r#gen.instructions.push(IrInstruction::I64Const(0));
            r#gen.instructions.push(IrInstruction::Return);
        }

        let mut local_groups = Vec::new();
        for ty in r#gen.locals {
            local_groups.push((1, ty));
        }

        let mut wasm_func = Function::new(local_groups);

        for instr in r#gen.instructions {
            wasm_func.instruction(&instr.to_wasm());
        }

        wasm_func.instruction(&Instruction::End);

        self.codes.function(&wasm_func);
    }
}

pub struct FuncGen {
    pub locals: Vec<ValType>,
    pub local_map: HashMap<String, u32>,
    // Instruction list for one function body.
```

```
    pub instructions: Vec<IrInstruction>,
    pub has_return: bool,
}
```

]

6.0.12 src/codegen/stmt.rs

[

```
use crate::codegen::ir::IrInstruction;
use crate::{
    ast::Stmt,
    codegen::{expr::emit_expr, module::FuncGen},
};
use std::collections::HashMap;
use wasm_encoder::*;

pub fn emit_stmt(stmt: &Stmt, r#gen: &mut FuncGen, funcs: &HashMap<String, (u32,
bool)>) {
    match stmt {
        Stmt::Let { name, value } => {
            let idx = r#gen.local_map.len() as u32;
            r#gen.locals.push(wasm_encoder::ValType::I64);
            r#gen.local_map.insert(name.clone(), idx);

            emit_expr(value, r#gen, funcs);
            r#gen.instructions.push(IrInstruction::LocalSet(idx));
        }

        Stmt::Expr(expr) => {
            // Drop only if this expression produced a stack value.
            let produced = emit_expr(expr, r#gen, funcs);
            if produced {
                r#gen.instructions.push(IrInstruction::Drop);
            }
        }

        Stmt::Return(expr_opt) => {
            match expr_opt {
                Some(expr) => {
                    emit_expr(expr, r#gen, funcs);
                }
                None => {
                    // If a return type exists, return a default zero value.
                    if r#gen.has_return {
                        r#gen.instructions.push(IrInstruction::I64Const(0));
                    }
                }
            }

            r#gen.instructions.push(IrInstruction::Return);
        }

        Stmt::If {
            cond,
            then_block,
```

```
            else_block,
        } => {
            emit_expr(cond, r#gen, funcs);
            // emit_expr leaves i64 values, while wasm `if` expects i32.
            // Convert i64 truthy/falsey into an i32 condition.
            r#gen.instructions.push(IrInstruction::I64Eqz);
            r#gen.instructions.push(IrInstruction::I32Eqz);
            r#gen.instructions.push(IrInstruction::If(BlockType::Empty));

            for s in then_block {
                emit_stmt(s, r#gen, funcs);
            }

            if !else_block.is_empty() {
                r#gen.instructions.push(IrInstruction::Else);
                for s in else_block {
                    emit_stmt(s, r#gen, funcs);
                }
            }

            r#gen.instructions.push(IrInstruction::End);
        }

        Stmt::While { cond, body } => {
            r#gen
                .instructions
                .push(IrInstruction::Block(BlockType::Empty));
            r#gen
                .instructions
                .push(IrInstruction::Loop(BlockType::Empty));

            emit_expr(cond, r#gen, funcs);
            r#gen.instructions.push(IrInstruction::I64Eqz);
            r#gen.instructions.push(IrInstruction::BrIf(1));

            for s in body {
                emit_stmt(s, r#gen, funcs);
            }

            r#gen.instructions.push(IrInstruction::Br(0));
            r#gen.instructions.push(IrInstruction::End);
            r#gen.instructions.push(IrInstruction::End);
        }
    }
}


]
```